COMP 302 Winter 2025 Problem Set 2

Problem 1: Factorial with Exception

Implement a function to compute factorial that raises an exception on negative input and explain why this function has type `int -> int`.

Problem 1 Solution

```
exception Negative_input

let rec factorial n =
  if n < 0 then raise Negative_input
  else if n = 0 then 1
  else n * factorial (n - 1)
```

- **Raising an Exception**: The inclusion of an exception for negative input (`raise Negative_input`) does not affect the type signature of the function. In OCaml, the `raise` function can technically be seen as having the type `'a -> 'b` because it can appear in any part of the function without affecting the expected return type. This is because `raise` never actually returns; it interrupts the normal flow of execution.

Problem 2: N-ary Tree Recursive Search

Implement a higher-order function to find a node that satisfies a predicate for an n-ary tree where each node has a list of subtrees.

Problem 2 Solution

```
exception NotFound

type 'a ntree = Empty | Node of 'a * 'a ntree list

let rec find p t = match t with
  | Empty -> raise NotFound
  | Node (x, ts) -> if p x then x else find_in_subtrees p ts

and find_in_subtrees p ts = match ts with
  | [] -> raise NotFound
  | t::ts -> try find p t with NotFound -> find_in_subtrees p ts
```

Problem 3: Recursive Search with Embedded Helper Function

Perform a recursive search on a tree similar to Problem 2 but embed a helper function using `let-in` to manage the recursive logic.

Problem 3 Solution

```
exception NotFound

type 'a ntree = Empty | Node of 'a * 'a ntree list

let find_with_helper p t =
  let rec find_internal t = match t with
    | Empty -> raise NotFound
    | Node (x, ts) ->
        if p x then x
        else find_in_subtrees ts
  and find_in_subtrees ts = match ts with
    | [] -> raise NotFound
    | t::rest ->
        try find_internal t with
        | NotFound -> find_in_subtrees rest
  in find_internal t
```

Problem 4: Recursive Search with Fold

Perform a recursive search on a tree like in Problem 2, but fold list recursion with tree recursion to streamline the recursive process.

Problem 4 Solution

```
exception NotFound

type 'a ntree = Empty | Node of 'a * 'a ntree list

let rec find_folded p t = match t with
  | Empty -> raise NotFound
  | Node (x, ts) when p x -> x
  | Node (_, ts) ->
      List.fold_left (fun acc subtree ->
        match acc with
        | exception NotFound -> find_folded p subtree
        | _ -> acc
      ) (raise NotFound) ts
```

Problem 5: Coffee System

You are tasked with implementing a coffee system Each customer can open an account to track their coffee purchases. For every 5th coffee purchase, the coffee should be free. Otherwise, the coffee costs a fixed price.

Requirements

1. Implement a function to open a new coffee account. Each account tracks the number of coffees a customer has purchased.
2. Implement a function to simulate the purchase of coffee. The function should return the price of the coffee (e.g., 2 units per coffee, and free on every 5th purchase).
3. Implement a function that returns the total number of coffees purchased on an account.

Function Signatures

```
val open_coffee_account : unit -> coffee_account
val buy_coffee : unit -> int
val get_purchases : unit -> int
```

Example Usage and Expected Output

```
let my_account = open_coffee_account ();;

my_account.buy_coffee ();;  (* Returns 2 *)
my_account.buy_coffee ();;  (* Returns 2 *)
my_account.buy_coffee ();;  (* Returns 2 *)
my_account.buy_coffee ();;  (* Returns 2 *)
my_account.buy_coffee ();;  (* Returns 0, indicating free coffee *)
my_account.get_purchases ();;  (* Returns 5 *)
```

Problem 5 Solution

```
type counter = {
  increment : unit -> unit;
  get_count : unit -> int;
}

let make_counter () =
  let count = ref 0 in {
  increment = (fun () -> count := !count + 1);
  get_count = (fun () -> !count);
}

type coffee_account = {
  buy_coffee : unit -> int;
```

```
  get_purchases : unit -> int;
}

let create_coffee_account () =
  let my_counter = make_counter () in
  let buy_coffee () =
    my_counter.increment ();
    if my_counter.get_count () mod 5 = 0 then 0
    else 2
  in
  let get_purchases () = my_counter.get_count () in
  { buy_coffee; get_purchases }
```

Problem 6: Simulating Mutable Lists

Simulate mutable lists using an immutable one.

```
type 'a mut_list = {
    append : 'a -> unit;
    drop : int -> unit;
    get_list : unit -> 'a list;
}

let make_mut_list l =
    let v = ref l in
    (* Implement *)
```

Example Usage and Expected Output

```
(* Creating a mutable list with initial elements *)
let myList = make_mut_list [1; 2; 3; 4; 5];;

(* Appending an element to the list *)
myList.append 6;;
myList.get_list ();;  (* Returns [1; 2; 3; 4; 5; 6] *)

(* Dropping the first element *)
myList.drop 2 ;
myList.get_list ();;  (* Returns [3; 4; 5; 6] *)
```

Problem 6 Solution

```
type 'a mut_list = {
    append : 'a -> unit;
    drop : int -> unit;
    get_list : unit -> 'a list;
}

let make_mut_list l =
    let v = ref l in
    let append x = v := !v @ [x]
    in
```

```
  let drop n =
    v := let rec drop_aux i lst = match lst with
      | [] -> []
      | _ :: tl -> if i > 0 then drop_aux (i - 1) tl else lst
    in drop_aux n !v
  in
  let get_list () = !v in
  { append; drop; get_list }
```